Submitted to PASP, 2012 Mar 1
Preprint typeset using LaTeX style emulateapj v. 5/2/11



# RAPID DEVELOPMENT OF INTERFEROMETRIC SOFTWARE USING MIRIAD AND PYTHON

Peter K. G. Williams, Casey J. Law, Geoffrey C. Bower
Department of Astronomy, B-20 Hearst Field Annex # 3411, University of California, Berkeley, CA 94720-3411, USA
*Submitted to PASP, 2012 Mar 1*

## ABSTRACT

New and upgraded radio interferometers produce data at massive rates and will require significant improvements in analysis techniques to reach their promised levels of performance in a routine manner. Until these techniques are fully developed, productivity and accessibility in scientific programming environments will be key bottlenecks in the pipeline leading from data-taking to research results. We present an open-source software package, miriad-python, that allows access to the MIRIAD interferometric reduction system in the Python programming language. The modular design of MIRIAD and the high productivity and accessibility of Python provide an excellent foundation for rapid development of interferometric software. Several other projects with similar goals exist and we describe them and compare miriad-python to them in detail. Along with an overview of the package design, we present sample code and applications, including the detection of millisecond astrophysical transients, determination and application of nonstandard calibration parameters, interactive data visualization, and a reduction pipeline using a directed acyclic graph dependency model analogous to that of the traditional Unix tool make. The key aspects of the miriad-python software project are documented. We find that miriad-python provides an extremely effective environment for prototyping new interferometric software, though certain existing packages provide far more infrastructure for some applications. While equivalent software written in compiled languages can be much faster than Python, there are many situations in which execution time is profitably exchanged for speed of development, code readability, accessibility to nonexpert programmers, quick interlinking with foreign software packages, and other virtues of the Python language.

*Subject headings:* Data Analysis and Techniques

## 1. INTRODUCTION

Advances in the fields of digital computing and commercial wireless communication have fueled an explosion of innovation in radio interferometry. New and upgraded facilities such as the Allen Telescope Array (ATA; Welch et al. 2009), the Low-Frequency Array (Kassim et al. 2004), the Precision Array for Probing the Epoch of Reionization (PAPER; Parsons et al. 2010), the Murchison Wide-Field Array (MWA; Lonsdale et al. 2009), the Karl G. Jansky Very Large Array (formerly EVLA; Perley et al. 2011), the Westerbork Synthesis Radio Telescope (WSRT) Apertif project (Verheijen et al. 2008), the Australian Square Kilometer Array Pathfinder (DeBoer et al. 2009), and MeerKAT (Jonas 2009) have sophisticated designs including large-number-of-small-dishes architectures, multipixel or phased-array feeds, wide bandwidths, and phased array substations. Several of these facilities are pathfinders for the proposed Square Kilometer Array (SKA; Carilli & Rawlings 2004). They all aim to push the limits of interferometric techniques to image large fields of view with unprecedented spatial, spectral, temporal, and polarimetric fidelity, at unprecedented data rates.

Achieving these goals will require substantial amounts of new software to process the data coming out of these facilities. The data rates present a challenge in and of themselves: SKA-class facilities will require exaflop-scale computing (Cornwell & Humphreys 2010). Another challenge is the development of the necessary new algorithms, which is certain to require extensive experimentation. Techniques already under investigation include $w$-projection (Cornwell et al. 2008), $A$-projection (Bhatnagar et al. 2008), multi-scale multi-frequency CLEAN (Rau & Cornwell 2011), delay/delay-rate filtering (Parsons & Backer 2009), space-alternating generalized expectation (SAGE) maximizing calibration (Kazemi et al. 2011), generalized measurement-equation-based instrumental modeling (Noordam & Smirnov 2010), scale-invariant rank detection of radiofrequency interference (RFI; Offringa et al. 2012), subspace-tracking RFI mitigation (Ellingson & Hampson 2002), visibility stacking (Hancock et al. 2011), bispectral pulse detection (Law & Bower 2011) and improved sourcefinding tools (Whiting 2012), to name a few. Until the next generation of algorithms is thoroughly explored, the quality of many results will be set not by the capabilities of observatory hardware but by the sophistication of the reduction pipeline that can be brought to bear before publication: *software-limited* science.

Although improvements in software development efficiency are always desirable, they're particular salient now as the next generation of radio interferometers comes online. We discuss the efficiency of a programming environment in terms of *productivity*, which we take to measure the amount of useful functionality that a programmer can implement per unit time, and *accessibility*, which measures how much effort it takes for non-experts to begin successfully working within the environment without hand-holding. Although many studies attempt to quantify these metrics (e.g., Petersen 2011, and references therein), our discussion will remain qualitative. These attributes are relevant in situations in which developer resources are constrained: a more productive environment allows individual developers to accomplish more, while a

pwilliams@astro.berkeley.edu

more accessible environment has a lower barrier to entry from informal contributors.

Interpreted, dynamic programming languages can provide such an environment. While several of these exist, the language Python[1] in particular has seen a broad uptake in the astronomical community over the past decade (e.g., Barrett & Bridgman 1999; Greenfield & White 2000; Blakeslee et al. 2003; Kettenis et al. 2006; Magee et al. 2007). It is intended to be easy to learn and offers conveniences including object-oriented programming, lambda expressions, exception handling, and an enormous software ecosystem, including interactive interpreters (e.g., IPython; Pérez & Granger 2007), visualization tools (e.g., matplotlib; Hunter 2007), file format interfaces (e.g., pyFITS; Barrett & Bridgman 1999), database interfaces (e.g., SQLite[2]), statistics routines, web service support, and so on. The existence of this ecosystem and the rapid rise in the popularity of Python in the astronomical community in particular speak to its productivity and accessibility.

The programming environments of "classic" astronomical software packages tend to be intimidating and frustrating in comparison. Two of the major traditional radio interferometric reduction packages, MIRIAD (Multichannel Image Reconstruction, Analysis, and Display; Sault et al. 1995) and AIPS (Astronomical Image Processing System; Greisen 2002), are largely implemented in FORTRAN-77, and in both cases the process to go from source code to usable installed executable code is complicated and fragile. In light of the clear need for improved interferometric algorithms, it should be no surprise that a variety of projects have sought to build on or replace these systems with more modern ones. Perhaps the most prominent such undertaking is the Common Astronomy Software Applications (CASA; McMullin et al. 2007), the successor to AIPS. Other packages include Obit (Cotton 2008b), AIPY (Astronomical Interferometry in Python[3]), MeqTrees (Noordam & Smirnov 2010), and several more narrowly-targeted binding layers (e.g., pyramid; Mehringer & Plante 2004).

To this list, we add miriad-python, a creatively-named package exposing MIRIAD tasks and subroutines in Python. Its design and philosophy are discussed (§2) and compared those of related projects (§3), including the ones mentioned above. We then describe the implementation of miriad-python (§4) and provide some very brief examples of its use (§5). Some applications in which it has been used are presented, including the detection of millisecond astrophysical transients, determination and application of nonstandard calibration parameters, interactive data visualization, and pipeline processing (§6). We document the nature of miriad-python as a software project (§7), discuss some of its performance characteristics (§8), and finally summarize (§9).

## 2. DESIGN CONSIDERATIONS IN miriad-python

The miriad-python project is intended to allow convenient access to MIRIAD tasks, datasets, and subroutines. Although it aims to ease the rapid development of new interferometric algorithms, it does not provide any nontrivial algorithms itself. As such, the focus of miriad-python is very narrow: it provides the best possible interfaces for access to MIRIAD infrastructure and is as flexible as possible regarding what is done with it. To use terminology dating back to at least the design of the X Windows system, it provides *mechanism* but not *policy* (Scheifler & Gettys 1986). We believe that the wide range of applications described in §6 is evidence of the power of this approach.

Although work on miriad-python was initially inspired by a practical desire for more efficient scripting of MIRIAD reductions, the MIRIAD package is a good fit to the larger goals of the miriad-python project. MIRIAD offers its wide variety of tools and algorithms in a modular task architecture, and its data formats are simple and efficient. In particular, it is fairly straightforward to (ab)use the streaming MIRIAD *u-v* data format for novel applications (see, e.g., AIPY, §3.2). Other packages are built upon MIRIAD for similar reasons (e.g., Teuben 2011; Pound & Teuben 2012). MIRIAD is routinely used to process data from facilities including the ATA, the Berkeley-Illinois-Maryland Association array (BIMA; Welch et al. 1996), the Combined Array for Research in Millimeter/submillimeter Astronomy (CARMA; Woody et al. 2004), the Australia Telescope Compact Array (ATCA), and the Submillimeter Array (SMA; Ho et al. 2004). MIRIAD is not a monolithic project: there are at least two nontrivially divergent codebases maintained for use with the ATCA and CARMA, with some sharing of modifications between the them. miriad-python is referenced to the CARMA MIRIAD codebase. Recent work on MIRIAD includes complete support for 64-bit file offsets and pointers, allowing imaging of arbitrarily large datasets; integration of the wcslib library (Calabretta 2011) for more comprehensive support of coordinate manipulations; and of course bug fixes, new features, and documentation improvements.

When creating a package such as miriad-python that interfaces with a lower-level one, one must decide how closely to hew to the APIs (application programming interfaces) of the original package. In miriad-python, the APIs have been significantly reworked: they are object-oriented and aim to take full advantage of builtin language features. When they meet the goal of providing mechanism rather than policy, substantial new features are implemented in the Python layer. One example is approximate cryptographic hashing of *u-v* datasets, which can be used to quickly and fairly robustly check for modifications to the data regardless of file modification times. (This is useful in dependency-tracking pipelines, cf. §6.4.) To write useful programs, miriad-python authors must be familiar with the semantics of MIRIAD datasets or tasks, and good knowledge of Python is helpful, but they do not need to understand the design of the MIRIAD subroutine library.

Another consideration for authors is whether to link to a separate installation of the lower-level software or to compile and install their own version of it, bundling a copy of its source code with their own. We have chosen to go the former route with miriad-python. Although miriad-python offers very different APIs and new features, we concieve of it as fundamentally a layer above MIRIAD and not a standalone package: if the user has customized their MIRIAD installation in some way, for instance, it's appropriate for miriad-python to reflect that customization, and not override it with its own copy of the subroutine

[1] http://python.org/
[2] http://sqlite.org/
[3] http://purl.org/net/pkgwpub/aipy

library. Another reason is that MIRIAD is still evolving, and tracking changes from an upstream codebase to a forked copy can be tedious and error-prone. Another is that miriad-python is a general-purpose interface to not just the MIRIAD subroutine library but its task collection, so the entire, substantial, MIRIAD codebase would need to be bundled. Finally, we expect that miriad-python users will be existing MIRIAD users who are likely to already have installed MIRIAD on their systems, so the convenience of bundled source code is lessened.

To touch on a broader philosophical issue, it may be argued that accessibility is not a desirable feature when it comes to interferometric software. Interferometric analysis tends to be subtle, and most people write buggy code (see §7.5). On the other hand, scientific research is a fundamentally creative and exploratory process often requiring new or improved techniques, and that is clearly the case in the domain of interferometry at present. The tension here is a variation on the well-explored "cathedral versus bazaar" theme first described by Raymond (1999). Considering the amount of algorithmic development necessary to fully exploit next-generation radio interferometers, we take the stance that the empowerment of interferometric software users is a good thing.

## 3. COMPARISONS TO RELATED PACKAGES

As alluded to in §1, there are several other efforts underway to provide developer-friendly environments for producing interferometric software. In this section we attempt to situate miriad-python among them. We consider "bindings", which do not provide substantial new algorithms, and "high-level" packages, which do.

### 3.1. *Bindings*

Several other packages expose MIRIAD functionality in the Python language, but we found them unsatisfactory. The pyramid package (Mehringer & Plante 2004), providing a module named Miriad, includes similar functionality to that of mirexec in a somewhat less structured manner. The modules mirlib (via the WSRT) and miriad-wrap both bind the MIRIAD subroutine library. Unlike miriad-python, they provide a fairly direct mapping to the MIRIAD subroutines and do not extensively "Pythonify" the API. These packages are less mature than miriad-python and do not appear to be under active development. None of them includes substantial documentation, high-level convenience features, or an integrated system for invoking both MIRIAD tasks and subroutines.

There also exist several other packages that appear to share the same motivations as miriad-python but involve different technologies. Other dynamic languages can be used. For instance, MIRIAD functionality can be accessed in Ruby[4] with the MIRIAD-Ruby[5] or mirdl[6] packages. The two packages, written by the same author, cover the link-versus-bundle tradeoff discussed above: MIRIAD-Ruby includes a portion of the MIRIAD source code, while mirdl links to a separate installation of the full MIRIAD libraries. Although it is beyond the scope of this paper to compare the merits of the languages, we note that Ruby has seen less uptake in the astronomical community than Python but is starting to make inroads (e.g., Lammers et al. 2002; Fuentes et al. 2007; Gutierrez-Kraybill et al. 2010).

Other packages build on top of different interferometry frameworks. ParselTongue (Kettenis et al. 2006) provides a Python interface to AIPS, building on top of parts of the Obit package (Cotton 2008b), described in the next subsection. The pyrap project[7] provides a Python interface to the core C++ support libraries of CASA.

In many cases, the choice of the appropriate binding will be constrained by external limitations in either the high-level language or the underlying interferometry package. Our reasons for preferring Python and MIRIAD, respectively, are sketched in §1 and §2. We emphasize the simplicity and efficiency of MIRIAD's data formats as being important factors in facilitating algorithmic experimentation. Another distinction between miriad-python and alternative bindings is its rate of development. Even in otherwise stable and mature packages, documentation improvements and subtle bugfixes tend to require fairly frequent modifications. In the calendar year of 2011, there were 5 commits to the mirdl version control system (VCS), 10 commits to the pyrap VCS, and 110 commits to that of miriad-python. The last known update to ParselTongue was in 2010 and the last known update to pyramid was in 2008. Finally, we believe that miriad-python acquits itself well when it comes to the subjective factors of API cleanliness, code readability, and code quality.

### 3.2. *High-Level Packages*

The projects described in this subsection are more ambitious than miriad-python and include significant functionality along with Python-based environments: they provide policy, as well as mechanism (cf. §2). While the packages described in the previous subsection are intended to provide suitable bases for development for a wide range of applications, the ones described below may not be appropriate for certain applications, depending on both technical and architectural factors. All of these packages are actively developed.

CASA (McMullin et al. 2007) is a substantial package including high-level Python interfaces to lower-level routines implemented in C++ and FORTRAN. The first iteration of CASA, known as AIPS++ (McMullin et al. 2004), included flexibility, ease of programming, and modifiability in its mission statement (AIPS++ Steering Committee 1993). We find that these are difficult to achieve in practice because most CASA functionality is implemented in fairly large C++ modules, with the built-in Python interfaces providing mostly coarse-grained access to data and algorithms; the combination of the core CASA libraries with pyrap is more friendly to Python development. One minor inconvenience is that CASA bundles its own Python interpreter, so that site-specific packages accessible to the system interpreter sometimes need to be rebuilt and reinstalled in order to be accessible to CASA's version. CASA's imager is quite flexible and can combine multiple advanced algorithms, for instance simultaneous multi-scale, multi-frequency deconvolution (Rau & Cornwell 2011).

Obit is billed as "a development environment for astronomical algorithms" (Cotton 2008b). Although many of

[4] http://www.ruby-lang.org/
[5] http://purl.org/net/pkgwpub/miriad-ruby
[6] http://purl.org/net/pkgwpub/mirdl
[7] http://purl.org/net/pkgwpub/pyrap

its fundamental features are general-purpose, Obit's main emphasis is radio astronomy, and in particular it includes several algorithms important to low-frequency radio interferometry. The package includes a set of base libraries implemented in C, a set of tasks built on top of them, and a Python layer for interacting with both of these. This layer is based on ParselTongue, although the two projects remain separate (W. D. Cotton, priv. comm.). Two aspects of the Obit design are of particular note. Firstly, Obit uses AIPS and FITS file formats for data storage, so it interoperates seamlessly with certain existing packages and developer effort need not be spent reimplementing existing tools. Secondly, the Obit libraries are architected to allow multithreaded execution, and this capability has been explored during Obit's development. In one example problem, the time taken to generate a complex image was reduced by a factor of 6.5 when the number of CPU cores used for computations was increased from one to twelve (Cotton & Perley 2010).

The AIPY system represents a sharper break from tradition than those previously mentioned. It is a standalone package implementing many advanced synthesis and calibration techniques in Python to meet the needs of the low-frequency dipole array PAPER. Its design emphasizes flexibility and is Python-centric: the primary interfaces are the Python ones, and code from preexisting packages such as HEALPix (Górski et al. 2005) and XEphem (Downey 2011) is liberally borrowed to provide fast, well-tested implementations of many features. AIPY can use the MIRIAD format for $u$-$v$ data storage, and hence includes an internal Python binding of the relevant portions of the MIRIAD subroutine library. A particularly interesting strength of AIPY is its infrastructure for creating maps of the whole sky at once.

The MeqTrees project (Noordam & Smirnov 2010) also uses Python to provide a productive environment for exploring new interferometric algorithms using the measurement equation formalism (Hamaker et al. 1996; Smirnov 2011). Its input/output (I/O) and astronomical infrastructure are built on the core CASA libraries and pyrap, with certain numerical routines implemented in C++. MeqTrees provides a substantial infrastructure for developing and applying sophisticated calibration methodologies with numerous visualization features. As an example of its power, MeqTrees has been used to generate a noise-limited radio astronomical image of the source 3C 147 with a dynamic range of $1.6 \times 10^6$ (Noordam & Smirnov 2010).

## 4. IMPLEMENTATION

miriad-python provides three Python modules: mirtask, mirexec, and miriad. The first of these uses C and FORTRAN-77 based extension modules to interface with the MIRIAD subroutine library, providing most of the low-level functionality that makes miriad-python useful. Numerical functionality is provided by NumPy[8] and linking with FORTRAN-77 is accomplished using f2py (Peterson 2009). The second module, mirexec, is Python-only and provides a uniform framework for executing MIRIAD tasks inside the Python language. The third, miriad, is Python-only and provides a simple abstraction for referencing and manipulating MIRIAD datasets, with hooks into mirtask and mirexec to provide convenient access to certain common operations. The three modules are loosely-coupled such that they do not actually depend on each other, although most nontrivial tasks end up using functionality from all three modules. The relationships of the APIs are diagrammed in Figure 1.

As mentioned above, the MIRIAD APIs are significantly reworked and expanded. In most cases, Python's runtime type information is used to automatically choose data types when performing lowlevel I/O. Because these types are important for task interoperability, they must be specified explicitly when writing new data. MIRIAD's internal errors are mapped into Python exceptions using a mechanism based on the C library functions setjmp and longjmp.

miriad-python also provides a program that can serve as a help viewer for either existing MIRIAD tasks or programs written in Python, in the latter case using standard Python "documentation strings" written in the standard MIRIAD documentation format. Tasks written in Python can thus seamlessly merge with traditional compiled MIRIAD tasks from the user standpoint.

## 5. EXAMPLE CODE

Although it is impractical to provide extensive code examples in this paper, in this section we give a few brief ones. These are necessarily simplistic and do not demonstrate the most interesting capabilities of miriad-python. As discussed below (§7.4), the miriad-python source distribution contains longer samples[9]. We assume a familiarity with typical MIRIAD usage and the Python language. Many valuable coding practices are ignored below for the sake of concision.

MIRIAD tasks can be executed with the mirexec module. The following example demonstrates the parallel execution of multiple tasks: the launch method starts a task process, returning a handle to it (a subclass of Python's subprocess.Popen), and the handle's wait method pauses the caller until the process completes. (For I/O-intensive operations as shown here, it may actually be faster to execute the tasks serially to avoid disk thrashing.)

```
import sys
from mirexec import TaskUVAver

prochandles = [ ]

for p in sys.argv[1:]:
    taskdesc = TaskUVAver (vis=p, out=p+".avg",
                           interval=10)
    prochandle = taskdesc.launch ()
    prochandles.append (prochandle)

for prochandle in prochandles:
    prochandle.wait ()
```

[8] http://numpy.scipy.org/

[9] Currently browseable online at https://github.com/pkgw/miriad-python/tree/master/examples.

The above example is intentionally verbose. To stimulate the reader's imagination, we provide the following variation that uses Python's "list comprehension" syntax and avoids the use of several local variables:

```
import sys
from mirexec import TaskUVAver

for ph in [TaskUVAver (vis=p, out=p+".avg", interval=10)
           .launch () for p in sys.argv[1:]]:
  ph.wait ()
```

The following example demonstrates how the "header variables" of MIRIAD datasets may be accessed using miriad-python. The variable vispath is some string giving the filesystem path of a MIRIAD $u$-$v$ dataset. As mentioned in §4, storage types of variables are detected automatically on read, but must be specified explicitly on write to enforce consistency with preexisting MIRIAD tasks.

```
import miriad, numpy as np

vispath = ...
ref = miriad.VisData (vispath)
handle = ref.open ("rw")
ncorr = handle.getScalarItem ("ncorr")
handle.setScalarItem ("mine", np.double, np.pi * ncorr)
```

The following example replaces the contents of an image with the magnitudes of its fast Fourier transform (FFT). This can be used, for instance, to recover the weights used in the imaging process from the dirty beam image. The variable impath is analogous to vispath. The value of the variable whichplane indicates that the first image plane should be selected.

```
import miriad
from numpy import abs, float32
from numpy.fft import fft2, fftshift, ifftshift

impath = ...
whichplane = [ ]
handle = miriad.ImData (impath).open ("rw")
d = handle.readPlane (whichplane).squeeze ()
d = abs (ifftshift (fft2 (fftshift (d))))
handle.writePlane (d.astype (float32), whichplane)
handle.close ()
```

Finally, this example inserts information about the synthesized beam shape of an image into a preexisting relational database.

```
import miriad, sqlite3

impath = ...
handle = miriad.ImData (impath).open ("rw")
dbpath = ...
db = sqlite3.connect (dbpath)

info = [impath]
for item in "bmaj bmin bpa".split ():
  info.append (handle.getScalarItem (item))

db.execute ("INSERT INTO data VALUES (?,?,?,?)", info)
db.commit ()
db.close ()
```

Such functionality would be difficult to achieve using FORTRAN. We emphasize that in the interests of simplicity and clarity, we have avoided some of the most important features of the Python language such as object orientation, first-class functions, and exception handling.

## 6. APPLICATIONS

The fast Python development cycle makes it relatively easy to produce new algorithms and tools for astronomical data analysis. In this section we present some example applications of miriad-python.

### 6.1. *Algorithms for Millisecond Transients*

We have investigated with new algorithms to study millisecond transients in the visibility domain using data from the ATA and VLA (Law et al. 2011; Law & Bower 2011). Since interferometers have not traditionally operated at this time scale, many interferometric packages lack features needed to work in this regime, e.g., the ability to dedisperse visibilities and high-resolution timestamps. We used miriad-python to develop new ways to visualize and search these unusual data streams for transient radio sources.

Figure 2 demonstrates imaging of millisecond pulses from the Crab pulsar (B0531+21) using a Python-based toolchain. We observed the Crab pulsar at a time resolution of 1.2 ms and frequencies between 720 to 800 MHz using the ATA (Law et al. 2011). At this time resolution, the dispersion introduced by the interstellar medium (ISM) delays the pulse arrival time by a few tens of milliseconds across the band. The upper panel of Figure 2 shows this effect in a spectrogram formed by summing visibilities, effectively forming a synthesized beam toward the Crab pulsar. The spectrogram shows that the dispersive delay follows a quadratic shape that has a slope consistent with that expected from the Crab pulsar. After identifying the time and dispersion of the pulse, we can image it to see if it is consistent with a point source in the location of the Crab pulsar. The $u$-$v$ input/output facilities of miriad-python are used to write a new dedispersed visibility dataset. The lower panel of Figure 2 shows an image made from visibilities during the pulse. The dispersive delay is large enough that it must be corrected in order to detect the source.

Low-level access to visibility data in miriad-python also makes it possible to experiment with new algorithms. We've used this capability to develop a technique to detect millisecond transients based on an interferometric closure quantity called the bispectrum (Cornwell 1987; Law & Bower 2011). The bispectrum is formed by multiplying three visibilities from baselines that form a closed loop. This product is sensitive to transients anywhere in the field of view, which makes it powerful for surveys. However, most software packages only use the bispectrum for calibration, so none have low-level access to functions for prototyping algorithms. Figure 3 shows millisecond light curves toward pulsar B0329+54 made with miriad-python. With direct access to dedispersed visibilities, we were able to compare traditional beamforming with the bispectrum technique. In this case, we can detect pulses for four of the five rotations of the pulsar during this observation. Being able to compare the techniques on the same pulses allowed us to show that the bispectrum responds more strongly than beamforming, confirming an unusual theoretical property of the bispectrum (Law & Bower 2011).

### 6.2. *Retroactive SEFD Calibration*

The ATA lacks online measurement of the system temperature ($T_{\rm sys}$) or system equivalent flux density (SEFD), information that is important for assessing data quality and thermal noise limits. We have written a task, calnoise, to assess per-antenna SEFDs after-the-fact using observations of a bandpass calibrator and the assumption that the variance across the channels of each spectral window is entirely thermal.

Each ATA antenna has two orthogonally linearly polarized feeds. The signal from each of these is identified as originating from a given antenna-polarization pairing ("antpol"). The SEFDs of the two feeds on a single antenna may differ appreciably, and all of the SEFDs may be time-variable. Rather than directly computing SEFDs, calnoise determines calibration coefficients relating SEFDs to the RMS value across the spectral window of each antpol's uncalibrated autocorrelation amplitude, a value we refer to as the RARA (raw autocorrelation RMS amplitude). While the value of this coefficient is assumed to be static over the course of an observing session, the derived SEFD value will vary proportionally with the RARA. Although the RARA is a function of both system noise and a gain factor, the time variation in the latter is small enough so that the derived SEFDs will be sufficiently accurate.

calnoise takes as an input a dataset with a gains (and optionally bandpass) table. The dataset is scanned through once, without applying the gains table, to accumulate RARA values for each timestamp and antpol. The dataset is then reread, with the gains being applied, to compute per-baseline SEFDs:

$$\mathrm{SEFD} = \frac{1}{2}\left(\sigma_r + \sigma_i\right)\eta\sqrt{2\Delta\nu\tau},$$

where $\sigma_r$ and $\sigma_i$ are the standard deviation in the real and imaginary parts, respectively, of the calibrated visibilities across the spectral window (thus in Jansky units), $\eta$ is a tabulated efficiency of the correlation (depending, e.g., on the number of bits used in digitization), $\Delta\nu$ is the channel width in the spectral window, and $\tau$ is the integration time of the record. Per-antenna calibration coefficients are then calculated using a least-squares fit of the baseline-based values assuming a simple geometric dependence:

$$\mathrm{SEFD}_{i,j,t} = \sqrt{c_i\mathrm{RARA}_{i,t}\cdot c_j\mathrm{RARA}_{j,t}},$$

where the $c_i$ are the desired calibration parameters. Outlier values (possibly due to RFI or hardware failures) are identified by user-specified clipping limits and iteratively removed from the fits. MIRIAD's on-the-fly calibration system does not support SEFD information, so the $c_i$ are then recorded in a textual table on disk; a companion task reads the table along with an input dataset and creates a new dataset with the SEFD information inserted. (For technical reasons, a constant $T_{\rm sys}$ is assumed and varying values of the MIRIAD variable jyperk are inserted into the dataset, where SEFD $= T_{\rm sys}\cdot$ jyperk and thus jyperk is related to the effective area of each receiving element.)

calnoise is written in Python and uses miriad-python to read the visibility data and for miscellaneous astronomical routines. It uses NumPy for its numerics and other Python libraries for optional plotting of diagnostic information, least-squares solving, storage of numerical metadata, and integration into the ATA pipeline described below (§6.4). The main implementation comprises ∼600 statement lines of code (i.e., excluding whitespace and comments). The chief benefits of using Python and miriad-python were quick creation of the task skeleton (thanks mainly to high-level data structures), rapid turnaround during refinement (thanks mainly to the lack of a compilation step), and easy investigation of algorithmic tweaks (thanks mainly to easy access to other libraries, e.g., only one line of code needed to interactively plot variables).

### 6.3. *Interactive u-v Data Visualization*

Data visualization is an important part of the development of any processing pipeline. This is especially true for radio interferometers, in which the data undergo complex transformations, are high-dimensioned, and closed-loop instrumental modeling is often a key aspect of the reduction. We have used miriad-python to implement a powerful *u-v* data visualizer, first described in Williams (2010). As shown in Fig 4, the main display is a dynamic spectrum of the visibilities on a baseline as a function of frequency and time. The user can quickly switch between different displays (real, imaginary, amplitude, phase), apply various processing steps on-the-fly (e.g., average, rephase), and navigate through the dataset. There is also substantial support for visualizing and creating data flags when necessary, although it is widely recognized that manual flagging of data is rapidly becoming an impossible task as data rates increase (Keating et al. 2010).

The use of Python is a key aspect to the *u-v* visualizer because it allows easy combination of the existing MIRIAD libraries with very different software, in this case the modern graphical toolkits Cairo[10] and GTK+[11]. Cairo provides routines for fast rendering of the gridded visibility data, while GTK+ provides a higher-level widget library that allows new user interactions to be implemented quickly. Achieving this functionality without the use of these (or similar) libraries would be a massive undertaking: Cairo and GTK+ comprise 240000 and 560000 lines, respectively, of well-tested, efficient C code. The main visualizer codebase, on the other hand, comprises only 2900 statement lines of Python.

### 6.4. *An ATA Reduction Pipeline*

Modern and next-generation radio observatories are expected to produce data at rates significantly higher than those of older facilities (e.g., Cornwell & Humphreys 2010). Under these conditions, rapid automated processing of data is changing from a luxury to a necessity (Keating et al. 2010). Python is often identified as a good system in which to construct pipelines to perform such processing, thanks to its amenability to implementing high-level application logic, revisability, and ability to glue together existing tools and libraries (Sanner 1999; Myers et al. 2007; Pérez et al. 2011). We have used miriad-python to implement a commensal observing system (Williams 2012) for the ATA Galactic Lightcurve and Transient Experiment (AGILITE; Williams et al., 2012, in prep.) and just such a pipeline for processing

[10] http://cairographics.org/
[11] http://gtk.org/

the data. In the latter case, the mirexec module is used extensively to invoke existing MIRIAD tasks and mirtask is used for low-level examination and manipulation of the data, often to accomplish tasks specific to ATA data reduction.

Of particular note is that the pipeline internally manages data flow through a directed acyclic graph model analogous to that of the traditional Unix tool make (Feldman 1979). While traditional shell scripting languages do not support the data structures necessary to conveniently implement such a design, doing so is easy in Python. The drPACS package (Teuben 2011) takes a similar approach but actually uses make for its underlying management. We find this approach to be unsatisfactory for two reasons. Firstly, make can only track dependencies between files on disk, and can only invoke tools via the Unix shell. This model maps very inefficiently onto certain aspects of typical data reduction workflows. Secondly, and more fundamentally, make checks whether steps need to be rerun by comparing file modification times rather than actual contents. This suffers from a variety of minor issues and the major issue that if a pipeline product is regenerated, all products downstream of it must also be regenerated, even if the rebuilt product did not actually change. In a data reduction pipeline the costs of this inefficiency can be severe. The tool described here avoids this problem by detecting modifications with cryptographic hashes as alluded to in §2.

## 7. miriad-python AS A SOFTWARE PROJECT

In this section we describe miriad-python as a software project. We also hope that this section may provide future authors a useful reference of issues that, in our opinion, are important to discuss when documenting any other software project.

### 7.1. *Intellectual Property Issues*

miriad-python is an open source project, almost entirely licensed under the GNU General Public License, version 3 or later. Small portions of it relating to the compilation process are licensed under different permissive licenses. As such, miriad-python is available for inspection and use by anyone for any purpose. The source code copyright is owned by the authors. miriad-python is not known to be subject to any software patents.

### 7.2. *Availability*

The source code is available for free download from the project website[12] either in the form of file archives or via the git[13] version control system. The git repository is also available on the website GitHub[14]. Thanks to the distributed nature of git, clones of either repository stand alone and contain the complete, cryptographically-verified revision history of the entire project[15]. Most installations of miriad-python track the git repository and so official releases are rare. When made, they are documented and linked to on the project website. The most recent release is version 0.6, made on 2011 April 28[16].

Installation instructions are beyond the scope of this paper and are provided on the website. We note, however, that miriad-python must be compiled against MIRIAD libraries from the CARMA codebase built with the autoconf-based build system.

### 7.3. *Development Model*

Development of miriad-python occurs in the git repository. The authors aim to conduct development in an open, welcoming fashion. Contributions from the community are encouraged via (e.g.) email or "pull requests" on the GitHub site. Based on the experiences of other community-based software projects, copyright assignment is not required for external contributions (see, for instance, the discussion in O'Mahony 2003).

The scale of the miriad-python project is such that other pieces of project coordination infrastructure such as mailing lists and a bug tracker are not currently deemed necessary. These will arise as the miriad-python community grows, and will be linked to on the project website.

### 7.4. *Documentation and Examples*

A moderately-complete manual to miriad-python is available on the website, providing both an API reference and somewhat higher-level guidance as to the intended usage of the package. The primary documentation format is HTML served over the web, but the documentation is generated using Sphinx[17], the official documentation system of the Python language, and in principle the documentation can be generated in several other output formats, including printable PDF files.

Besides the brief examples in §5, the miriad-python source distribution contains a small set of longer samples implementing features as both standalone scripts and importable modules. Exercised features include reading and writing of UV data, executing MIRIAD tasks with mirtask, manipulating dataset header items, reading gains tables, and integrating into the MIRIAD documentation system.

### 7.5. *Quality Assurance*

miriad-python is in line with the vast majority of scientific software in that, unfortunately, it has virtually no quality-assurance systems in place beyond the fact that it is exercised in day-to-day use. In particular, it has no automated test suite. Code changes in diff format are reviewed before being committed to the repository, hopefully preventing unintended changes from being integrated into the source tree. An automatic test framework is planned, using short unit tests of the miriad-python APIs as well as more involved tests using sample MIRIAD datasets either generated on-the-fly (with uvgen) or optionally downloaded from the project website.

[12] http://purl.org/net/pkgwpub/miriad-python. This link is a "permanent URL" providing a durable, reroutable link suitable for inclusion in the academic literature. The authors encourage the use of this URL and not its current, possibly-ephemeral destination when linking to the miriad-python website and related pages. See http://purl.org/ for more information.

[13] http://git-scm.com/

[14] Currently https://github.com/pkgw/miriad-python/, but the canonical link may be found via the project website.

[15] The SHA1 checksum of the version of miriad-python described in this paper is aff0a3cb4d47d030426cf9e36475e4bc9ae1816f. The secure architecture of git ensures that a valid commit of this checksum is guaranteed to correspond to the exact source code described in this work.

[16] The corresponding git commit is acbfa731c5da07c0270b03e9e93c11f4c9b7a4d4.

[17] http://sphinx.pocoo.org

Given time constraints and the potential payoffs it is certainly not hard to understand why scientific software so often lacks systematic testing infrastructure. As a point of reference, testing can approach half the cost of a piece of software in the corporate context (Machado et al. 2010). We emphasize, however, that software defects are extraordinarily common — about one for every *twenty* statement lines of code for undisciplined coding (Shull et al. 2002) — and that even rudimentary testing can capture many of these defects (e.g., Binder 1996). Shull et al. (2002) present an excellent summary of the prevalence and impact of defects in typical software systems.

## 8. PERFORMANCE CONSIDERATIONS

Python, being a dynamic interpreted language, is not optimal for achieving raw numerical throughput. Indeed, the *raison d'être* of a tool such as miriad-python is that in many cases numerical throughput can profitably be traded for other desirables such as code readability, programmer time savings, and ease of integration with existing codebases. In our experience, data-processing algorithms written in Python are almost always "fast enough," even for fairly complex algorithms.

### 8.1. *Serial Tests*

To give a very rough quantification of the overhead of using Python compared to FORTRAN-77, we wrote two versions of a simple task that iterates through a MIRIAD *u-v* dataset and computes the RMS value of the unflagged visibilities in each record. One version was written in Python and one in FORTRAN-77, the latter being compiled with GNU gfortran version 4.4.1. For a 0.3 GB visibility dataset containing 2,812,072 *u-v* records of 16 channels each, the Python version ran about 21 times slower than the FORTRAN-77 version (63 s vs 3 s) on our test system, a machine running a quad-core AMD Opteron 1385 CPU at 2.7 GHz. (Tests were run repeatedly with no effort to flush filesystem caches, so performance was CPU-limited.) For a 1.6 GB dataset containing 406,260 *u-v* records of 1,024 channels each, the Python version ran about 4 times slower (26 s vs 7 s). These results suggest that there is indeed a nonnegligible overhead to the Python interpreter, although its significance can be highly dependent on the structure of the input data. The performance of the Python version in the 1,024-channel case is relatively good because relatively more work is done inside a few vectorized NumPy function calls; manually iterating over each channel in Python increases the runtime by a factor of ∼90. For cases where most of the work is performed inside a few vectorized NumPy functions, the overhead to using Python can be fairly small.

In most software, the majority of the execution time is spent in only a small portion of the code. For those cases where serial throughput is a limitation, large improvements can often be achieved for low effort by identifying the most-executed codepaths and porting their particular implementations to a compiled language. This strategy is adopted by several of the projects discussed in §3. In some instances, it may be possible and profitable to generate speed-critical code on-the-fly using a compiler library such as the Low-Level Virtual Machine (Lattner & Adve 2004), avoiding the use of customized precompiled modules altogether. In other cases, it may make sense to prototype an algorithm in Python and then develop a production implementation in a compiled language. Of course, there will always exist problems for which Python is an inappropriate tool, but in our experience the combination of Python and a few basic numerical libraries is sufficient to address a wide range of challenges.

### 8.2. *Parallelization*

It is clear that parallelized processing will play a fundamental role in the reduction of future interferometric datasets. There is a special opportunity for next-generation interferometric processing packages to establish themselves in this space because the existing packages were generally built purely for serial processing.

There are several classes of parallelization that may be considered. For some applications, multiple independent reduction processes may be executed simultaneously on a single machine or on a cluster with no intra-process communication or synchronization. In this "embarrassingly parallel" case, a language such as Python may ease the development of both the reduction processes and the system to manage the overall processing job. Indeed, there are several preexisting Python frameworks for constructing parallel applications, e.g. Parallel Python[18].

Other parallel applications can be implemented via multiple independent processes that communicate and synchronize. Such applications are often implemented with the standard Message Passing Interface (MPI; Walker 1994), which is accessible in Python via several different toolkits (e.g., mpi4py; Dalcin et al. 2008). Somewhat surprisingly, MPI-based parallel Python programs with core numerical routines written in a compiled language can perform as well as parallel programs written purely in a compiled language (Cai et al. 2005). The comparative ease of writing the driving logic of such a program in Python versus compiled languages makes this an intriguing model for parallel algorithmic development.

For applications that require more intensive data-sharing, one may wish to parallelize MIRIAD operations within a single process by using multithreading. Unfortunately, Python's "global interpreter lock" prevents multiple threads from actually executing concurrently. (In many cases, naïvely multithreading a Python program makes it much slower!) In the particular case of miriad-python, extreme care would also be needed to synchronize access to the MIRIAD subroutine libraries, which are not threadsafe and maintain substantial shared state. Cotton (2008a) describes some of the challenges and successes encountered in tackling this kind of problem in Obit. The recommended system for threaded-style computation in Python is the multiprocessing module, which provides an infrastructure for launching concurrent Python subprocesses and communicating objects between parent and child. In the context of this work, multiprocessing is a more limited, Python-specific variant of the standard MPI approach. For certain I/O-limited applications, multithreaded Python code can obtain a performance gain above single-threaded code, but the difficulties of serializing access to the MIRIAD libraries remain.

Finally, certain classes of problems are well-suited to the highly-parallel capabilities of GPUs (graphics processing units). Current uses of GPUs in interferometric

[18] http://www.parallelpython.com/

applications include pulse dedispersion (e.g., Magro et al. 2011; Straten & Bailes 2011), RFI mitigation (Ait-Allal et al. 2012), cross-correlation (Clark et al. 2011), and visualization (Hassan et al. 2011). Python code can take advantage of GPUs using frameworks such as PyCUDA (Klöckner et al. 2012). The nature of these examples suggest that the algorithms most suited to GPU processing would not likely need to build on the MIRIAD/miriad-python infrastructure, except perhaps to use MIRIAD data formats for I/O.

Regardless of the kind of parallelization in question, many interferometric algorithms are I/O-intensive. The options for improving the I/O bandwidth of MIRIAD/miriad-python are somewhat limited due to the lack of built-in support for parallel I/O (e.g., Thakur et al. 1999) in the MIRIAD I/O subroutines. With investment in the appropriate hardware, speedups can be achieved using transparently parallel filesystems (e.g., Lustre[19] or PVFS; Carns et al. 2000). Significant gains in effective I/O bandwidth may alternatively be achieved with commodity hardware by spreading data between many independent processing nodes if the application permits partitioning of the data (e.g., by spectral channel or pointing direction).

## 9. SUMMARY

The new generation of interferometric hardware demands a new generation of interferometric software implementing a new generation of algorithms. Fortunately, there's a wide variety of projects aiming to provide the infrastructure needed to develop these tools as well as going ahead and implementing them.

miriad-python is one of these projects. It is "broad" rather than "tall": it provides a wide range of APIs for accessing MIRIAD tasks and data, but does not provide its own algorithms built on top of those APIs. While certain applications will be best matched to the facilities provided by the higher-level packages Obit, AIPY, or MeqTrees, miriad-python is a good foundation on which to build those that are not, especially if one wishes to take advantage of the simple and efficient MIRIAD $u$-$v$ data format. We actively encourage contributions to miriad-python development and are interested in supporting new miriad-python applications.

It seems likely that SKA-scale interferometric software will be only tangentially, if at all, related to any of the packages that exist today, and it is unclear if humans will "reduce" SKA data in any meaningful way given the rates involved. If the SKA is to meet both its goals and its schedule, however, the techniques needed to conquer exascale data rates must be discovered and prototyped, piece by piece, using today's software packages. Experience suggests that the innovations needed to solve these kinds of massive challenges do not come out of one organization; instead, wide-ranging and vibrant experimentation by a broad community of contributors will be essential.

The authors thank W. D. Cotton for helpful discussions. miriad-python is built upon the open-source Python language as well as an extensive foundation of other open-source and Free Software packages. It could not exist without the work of the innumerable people whose contributions have helped build this infrastructure. Research with the ATA is supported by the Paul G. Allen Family Foundation, the National Science Foundation, the US Naval Observatory, and other public and private donors. This research has made use of NASA's Astrophysics Data System.

[19] http://lustre.org/

## REFERENCES

AIPS++ Steering Committee. 1993, AIPS++ Memo Series, #110
Ait-Allal, D., Weber, R., Dumez-Viou, C., Cognard, I., & Theureau, G. 2012, Comptes Rendus Physique, 13, 80
Barrett, P. E., & Bridgman, W. T. 1999, in ASP Conf. Ser. 172, Astronomical Data Analysis Software and Systems VIII, ed. D. M. Mehringer, R. L. Plante, & D. A. Roberts (San Francisco, CA: ASP), 483
Bhatnagar, S., Cornwell, T. J., Golap, K., & Uson, J. M. 2008, A&A, 487, 419
Binder, R. V. 1996, Software: Testing, Verification, and Reliability, 6, 125
Blakeslee, J. P., Anderson, K. R., Meurer, G. R., Benítez, N., & Magee, D. 2003, in ASP Conf. Ser. 295, Astronomical Data Analysis Software and Systems XII, ed. H. E. Payne, R. I. Jedrzejewski, & R. N. Hook (San Francisco, CA: ASP), 257
Cai, X., Langtangen, H. P., & Moe, H. 2005, Scientific Programming, 13, 31
Calabretta, M. R. 2011, in Astrophysics Source Code Library, record ascl:1108.003
Carilli, C. L., & Rawlings, S. 2004, New Astronomy Review, 48, 979
Carns, P. H., Ligon, W. B., Ross, R. B., & Thakur, R. 2000, in Proceedings of the 4th annual Linux Showcase & Conference, Vol 4 (Berkeley, CA, USA: USENIX Association), 28
Clark, M. A., Plante, P. C. L., & Greenhill, L. J. 2011, International Journal of High Performance Computing Applications, submitted, arXiv:1107.4264
Cornwell, T., & Humphreys, B. 2010, SKA Memo Series, #128
Cornwell, T. J. 1987, A&A, 180, 269
Cornwell, T. J., Golap, K., & Bhatnagar, S. 2008, IEEE Journal of Selected Topics in Signal Processing, 2, 647
Cotton, W. D. 2008a, Obit Development Memo Series, #1
—. 2008b, PASP, 120, 439
Cotton, W. D., & Perley, R. 2010, Obit Development Memo Series, #21
Dalcin, L., Paz, R., Storti, M., & Delia, J. 2008, Journal of Parallel and Distributed Computing, 68, 655
DeBoer, D. R., Gough, R. G., Bunton, J. D., et al. 2009, Proc. IEEE, 97, 1507
Downey, E. C. 2011, in Astrophysics Source Code Library, record ascl:1112.013
Ellingson, S. W., & Hampson, G. A. 2002, IEEE Transactions on Antennas and Propagation, 50, 25
Feldman, S. I. 1979, Software: Practice and Experience, 9, 255
Fuentes, E., Miller, C. J., & Gasson, D. 2007, in ASP Conf. Ser. 376, Astronomical Data Analysis Software and Systems XVI, ed. R. A. Shaw, F. Hill, & D. J. Bell (San Francisco, CA: ASP), 579
Górski, K. M., Hivon, E., Banday, A. J., et al. 2005, ApJ, 622, 759
Greenfield, P., & White, R. L. 2000, in ASP Conf. Ser. 216, Astronomical Data Analysis Software and Systems IX, ed. N. Manset, C. Veillet, & D. Crabtree (San Francisco, CA: ASP), 59
Greisen, E. W. 2002, Astrophysics and Space Science Library, 285, 109
Gutierrez-Kraybill, C., Keating, G. K., MacMahon, D., et al. 2010, Proc. SPIE, 7740, 77400Z
Hamaker, J. P., Bregman, J. D., & Sault, R. J. 1996, A&AS, 117, 137
Hancock, P. P., Gaensler, B. M., & Murphy, T. 2011, ApJL, 735, L35
Hassan, A. H., Fluke, C. J., & Barnes, D. G. 2011, New Astronomy, 16, 100
Ho, P. T. P., Moran, J. M., & Lo, K. Y. 2004, ApJL, 616, L1
Hunter, J. D. 2007, Computing in Science and Engineering, 9, 90
Jonas, J. L. 2009, Proc. IEEE, 97, 1522
Kassim, N. E., Lazio, T. J. W., Ray, P. S., et al. 2004, Planet. Space Sci., 52, 1343

Kazemi, S., Yatawatta, S., Zaroubi, S., et al. 2011, MNRAS, 414, 1656
Keating, G. K., Barott, W. C., & Wright, M. 2010, Proc. SPIE, 7740, 774016
Kettenis, M., van Langevelde, H. J., Reynolds, C., & Cotton, B. 2006, in ASP Conf. Ser. 351, Astronomical Data Analysis Software and Systems XV, ed. C. Gabriel, C. Arviset, D. Ponz, & S. Enrique (San Francisco, CA: ASP), 497
Klöckner, A., Pinto, N., Lee, Y., et al. 2012, Parallel Computing, 38, 157
Lammers, U., Beijersbergen, M., Thomas, M., & Vacanti, G. 2002, in ASP Conf. Ser. 281, Astronomical Data Analysis Software and Systems XI, ed. D. A. Bohlender, D. Durand, & T. H. Handley (San Francisco, CA: ASP), 269
Lattner, C., & Adve, V. 2004, in IEEE/ACM International Symposium on Code Generation and Optimization (Los Alamitos, CA: IEEE), 75
Law, C. J., & Bower, G. C. 2011, ApJ, in press, arXiv:1112.0308
Law, C. J., Jones, G., Backer, D. C., et al. 2011, ApJ, 742, 12
Lonsdale, C. J., Cappallo, R. J., Morales, M. F., et al. 2009, Proc. IEEE, 97, 1497
Machado, P., Vincenzi, A., & Maldonado, J. C. 2010, Lecture Notes in Computer Science, 6153, 1
Magee, D. K., Bouwens, R. J., & Illingworth, G. D. 2007, in ASP Conf. Ser. 376, Astronomical Data Analysis Software and Systems XVI, ed. R. A. Shaw, F. Hill, & D. J. Bell (San Francisco, CA: ASP), 261
Magro, A., Karastergiou, A., Salvini, S., et al. 2011, MNRAS, 417, 2642
McMullin, J. P., Golap, K., & Myers, S. T. 2004, in ASP Conf. Ser. 314, Astronomical Data Analysis Software and Systems XIII, ed. F. Ochsenbein, M. G. Allen, & D. Egret (San Francisco, CA: ASP), 468
McMullin, J. P., Waters, B., Schiebel, D., Young, W., & Golap, K. 2007, in ASP Conf. Ser. 376, Astronomical Data Analysis Software and Systems XVI, ed. R. A. Shaw, F. Hill, & D. J. Bell (San Francisco, CA: ASP), 127
Mehringer, D. M., & Plante, R. 2004, in ASP Conf. Ser. 314, Astronomical Data Analysis Software and Systems XIII, ed. F. Ochsenbein, M. G. Allen, & D. Egret (San Francisco, CA: ASP), 42
Myers, C. R., Gutenkunst, R. N., & Sethna, J. P. 2007, Computing in Science and Engineering, 9, 34
Noordam, J. E., & Smirnov, O. M. 2010, A&A, 524, A61
Offringa, A. R., van de Gronde, J. J., & Roerdink, J. B. T. M. 2012, A&A, in press, arXiv:1201.3364
O'Mahony, S. 2003, Research Policy, 32, 1179
Parsons, A. R., & Backer, D. C. 2009, AJ, 138, 219
Parsons, A. R., Backer, D. C., Foster, G. S., et al. 2010, AJ, 139, 1468
Pérez, F., & Granger, B. E. 2007, Computing in Science and Engineering, 9, 21
Pérez, F., Granger, B. E., & Hunter, J. D. 2011, Computing in Science and Engineering, 13, 13
Perley, R. A., Chandler, C. J., Butler, B. J., & Wrobel, J. M. 2011, ApJL, 739, L1
Petersen, K. 2011, Information and Software Technology, 53, 317
Peterson, P. 2009, International Journal of Computational Science and Engineering, 4, 296
Pound, M. W., & Teuben, P. 2012, arXiv:1202.1030
Rau, U., & Cornwell, T. J. 2011, A&A, 532, A71
Raymond, E. 1999, Knowledge, Technology and Policy, 12, 23
Sanner, M. F. 1999, Journal of Molecular Graphics and Modelling, 17, 57
Sault, R. J., Teuben, P. J., & Wright, M. C. H. 1995, in ASP Conf. Ser. 77, Astronomical Data Analysis Software and Systems IV, ed. R. A. Shaw, H. E. Payne, & J. J. E. Hayes (San Francisco, CA: ASP), 433
Scheifler, R. W., & Gettys, J. 1986, ACM Trans. Graph., 5, 79
Shull, F., Basili, V., Boehm, B., et al. 2002, in Proceedings of the Eighth IEEE Symposium on Software Metrics (Ottawa, Canada: IEEE), 249
Smirnov, O. M. 2011, A&A, 527, A106
Straten, W. V., & Bailes, M. 2011, PASA, 28, 1
Teuben, P. 2011, in ASP Conf. Ser. 442, Astronomical Data Analysis Software and Systems XX, ed. I. N. Evans, A. Accomazzi, D. J. Mink, & A. H. Rots (San Francisco, CA: ASP), 533
Thakur, R., Gropp, W., & Lusk, E. 1999, in Frontiers '99: The Seventh Symposium on the Frontiers of Massively Parallel Computation (Los Alamitos, CA, USA: IEEE), 182
Verheijen, M. A. W., Oosterloo, T. A., van Cappellen, W. A., et al. 2008, in AIP Conf. Ser. 1035, The Evolution of Galaxies Through the Neutral Hydrogen Window, ed. R. Minchin & E. Momjian (Melville, NY: AIP), 265
Walker, D. W. 1994, Parallel Computing, 20, 657
Welch, J., Backer, D., Blitz, L., et al. 2009, Proc. IEEE, 97, 1438
Welch, W. J., Thornton, D. D., Plambeck, R. L., et al. 1996, PASP, 108, 93
Whiting, M. T. 2012, MNRAS, in press, arXiv:1201.2710
Williams, P. 2010, in Proceedings of Science, RFI Mitigation Workshop, PoS(RFI2010)004
Williams, P. K. G. 2012, Allen Telescope Array Memo Series, #89
Woody, D. P., Beasley, A. J., Bolatto, A. D., et al. 2004, in SPIE Conf. Ser. 5498, ed. C. M. Bradford, P. A. R. Ade, J. E. Aguirre, J. J. Bock, M. Dragovan, L. Duband, L. Earle, J. Glenn, H. Matsuhara, B. J. Naylor, H. T. Nguyen, M. Yun, & J. Zmuidzinas, 30

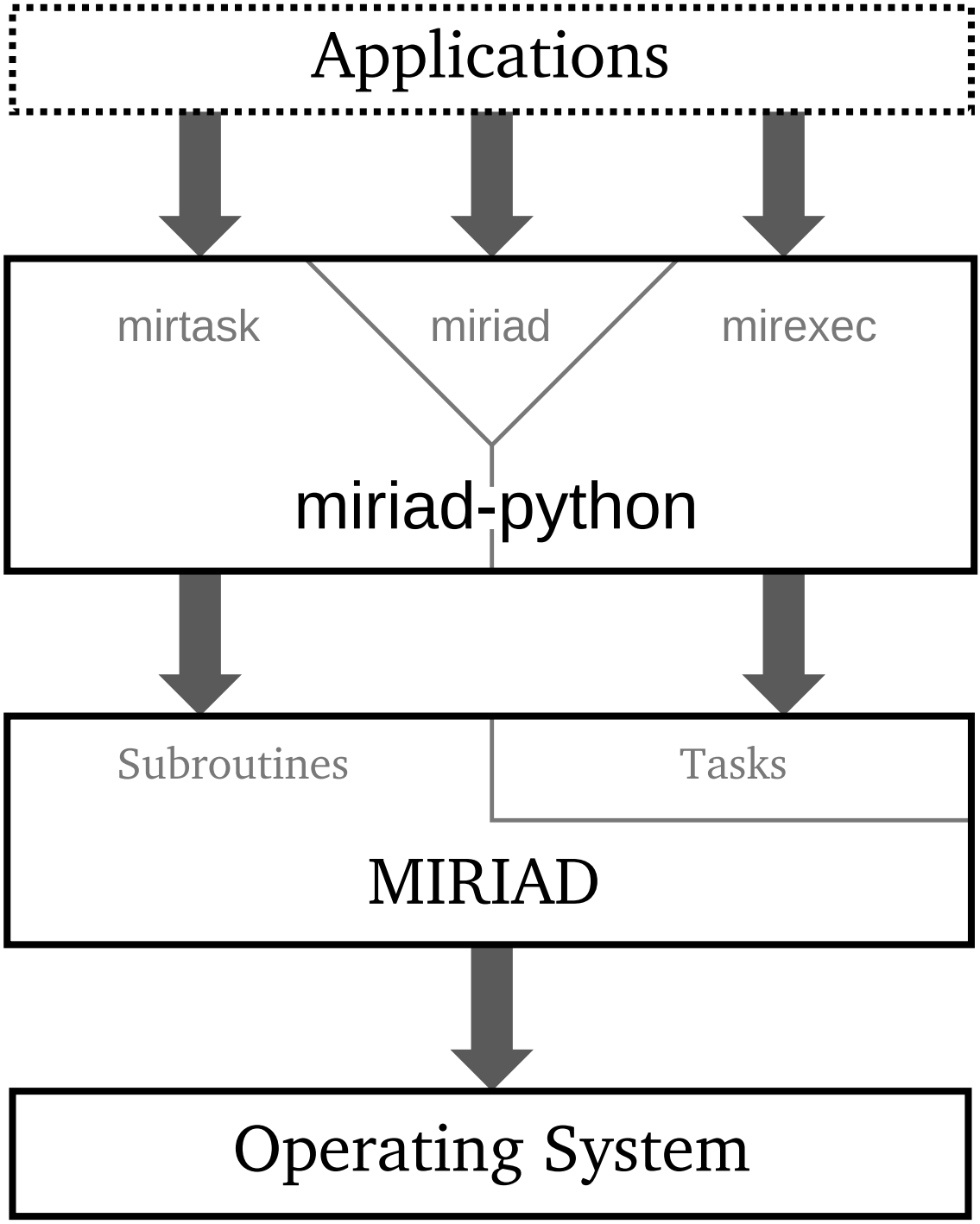


**Figure 1.** Diagram of the relationships between different components of the miriad-python system. Applications use the three miriad-python modules, which in turn access the tasks and subroutines in MIRIAD, which interact with the base operating system. Both applications and miriad-python use the NumPy module (not depicted) for numerical array operations.

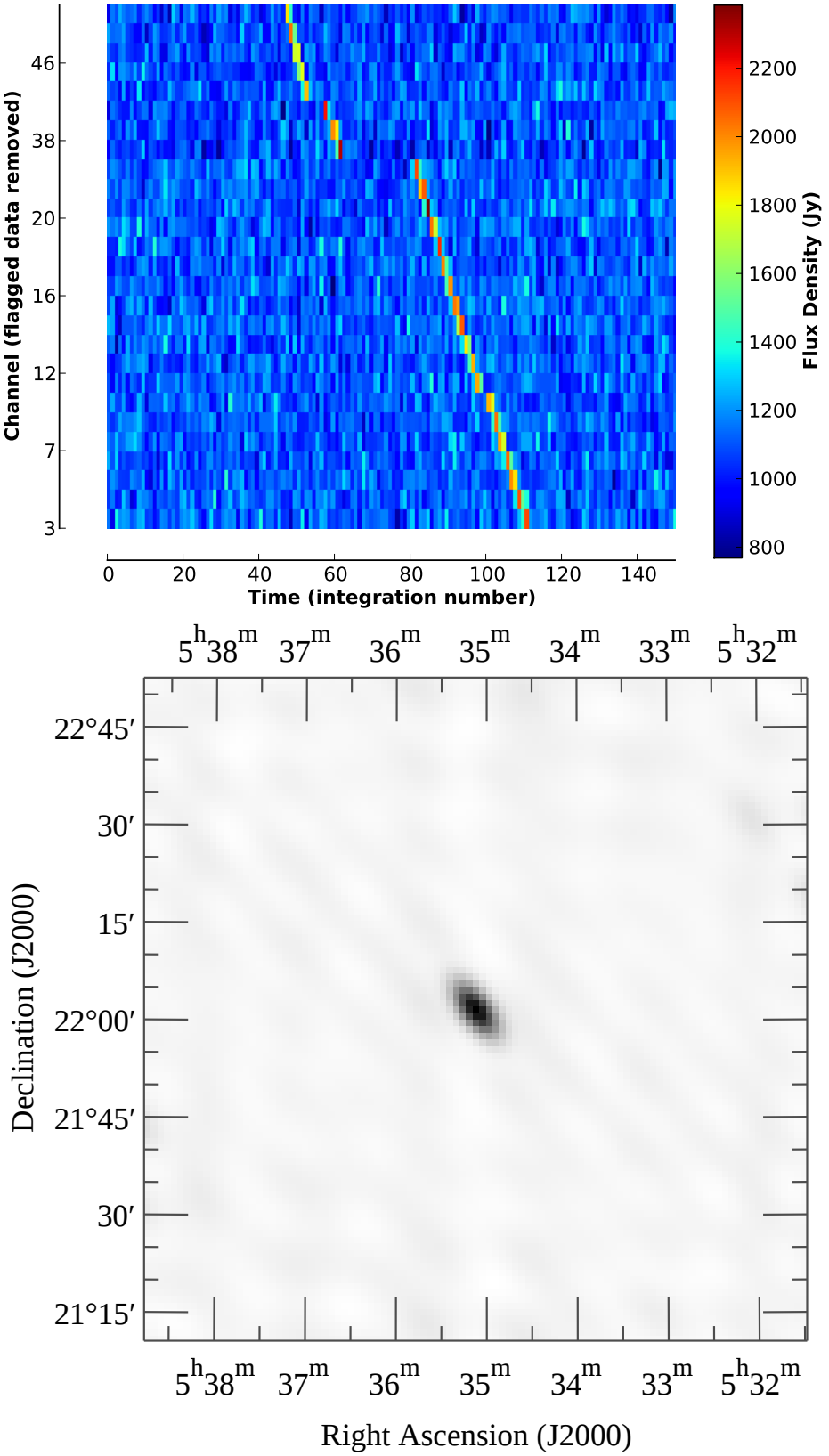


**Figure 2.** *Upper panel:* Spectrogram of a pulse from the Crab pulsar, summing visibilities coherently using the known location of the emitter. Dispersion due to the ISM causes pulse arrival times to vary with frequency (channel number). Apparent discontinuities in the pulse are due to flagged channels not being rendered (note the gaps in the left axis values). *Lower panel:* An image of the pulsar combining all visibilities in the observation, applying a correction for the dispersive time delay. The grayscale is linear from -42 Jy (white) to 1189 Jy (black). See §6.1 and Law et al. (2011).

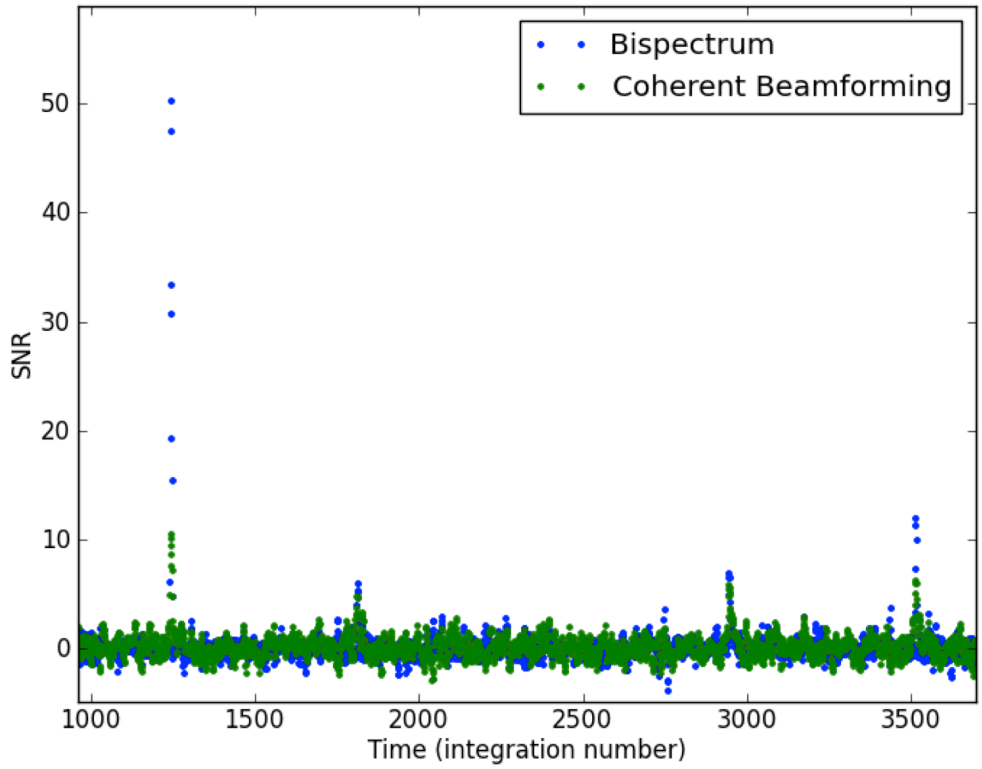


**Figure 3.** Comparison of two methods for detecting millisecond astrophysical pulses, one using the bispectrum and one using traditional beamforming, both implemented in Python. Each method is applied to a visibility data stream containing five pulses from the bright pulsar B0329+54 and the signal-to-noise ratio (SNR) of each detection is plotted. The bispectrum-based method can be more effective (higher SNR detections for the same data) than coherent beamforming and is sensitive to pulses coming from any direction. See §6.1 and Law & Bower (2011).

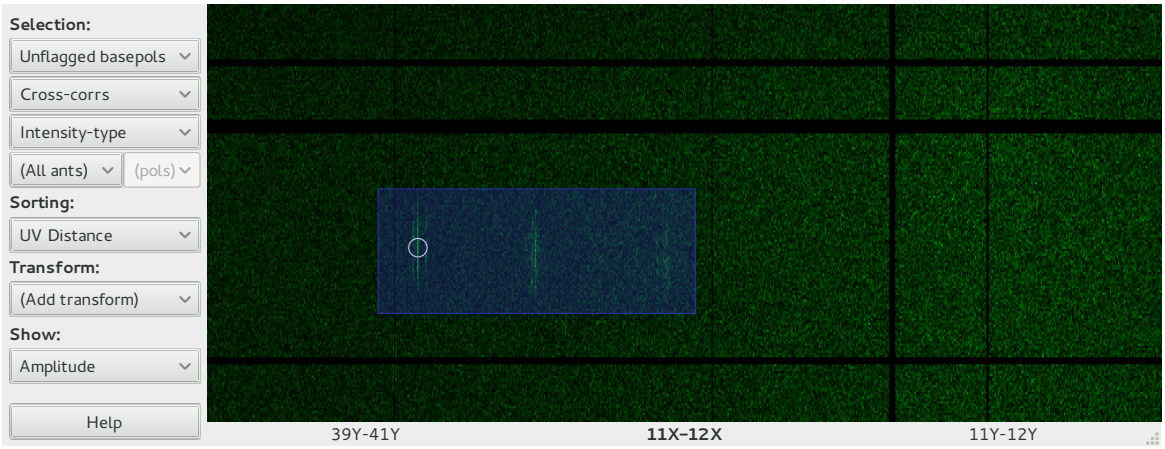


**Figure 4.** Sample screenshot of the *u-v* data visualizer described in §6.3. The main graphical display is a dynamic spectrum of amplitude on one baseline as a function of frequency (horizontal axis) and time (vertical axis). The left-hand panel provides controls for filtering and transforming the data. Python bindings to well-established graphical toolkits allow the data to be displayed in an attractive and responsive user interface.